Integration of a Solid Modeler with a Feature-oriented Database

Tang S. –H.[1], Ma Y. –S.[2] and Chen G.[3]

[1] DRC, School of MAE, Nanyang Technological University, Singapore, pg02104852@ntu.edu.sg
[2] School of MAE, Nanyang Technological University, Singapore, mysma@ntu.edu.sg
[3] CADCAM Lab, School of MAE, Nanyang Technological University, Singapore, pg02198079@ntu.edu.sg

ABSTRACT

Feature-oriented databases are ideal to support Web enabled collaborative engineering services. To implement a feature-oriented database, a solid modeler is incorporated to provide lower-level modeling services. In this paper, based on a generic feature definition for different applications and with a four-layer information integration infrastructure, the EXPRESS-schemas for the generic feature representation and constraint definition are given. The integration of solid modeler with feature-oriented database can be achieved by: Mapping from EXPRESS-defined feature model to the runtime solid modeler data structure and to the targeted database schema; Proposing modeler-based generic algorithms for feature validation and data manipulation in the database; and providing modeler-supported label-based approach for feature model re-evaluation.

Keywords: Web services, Feature modeling; Feature-oriented database; EXPRESS; Collaborative engineering

1. INTRODUCTION

Traditionally, solid modelers are designed to support CAD systems, but almost all existing CAD systems are based on files for their repositories. Such design is no longer adequate for networked engineering environment. The evolvement of Web services demands reliable databases which can support the fine grain information repository. In the previous work [12], a four-layer information integration infrastructure is proposed based on the architecture of a feature-oriented database. Ideally, it will enable information sharing among CAx applications by using the unified feature model [5] in the Entire Product Model (EPM), and allows the manipulation of application-specific information with sub-models. However, the method to incorporate a solid modeler into the system for providing low-level geometrical modeling services remained as a major question for research. Martino et al [9] proposed an intermediate geometry modeler to integrate design and other engineering processes with a combined approach of “design-by-feature” and “feature recognition”. Bidarra [2, 3] and Bronsvoort [4] proposed a semantic feature model by incorporating ACIS into webSPIFF, a web-based collaborative system. However, integration with a database is not mentioned in above-mentioned research because it is not clear whether they have managed product data in files or with a database. Kim et al [7] describes an interface (OpenDIS) for the integration of a geometrical modeling kernel (OpenCascade) and a STEP database (ObjectStore). However, the proposed system does not support features. This research work is to investigate the mechanisms to integrate a solid modeler with a feature-oriented database, such that multi-application collaboration can be realized over the Web.

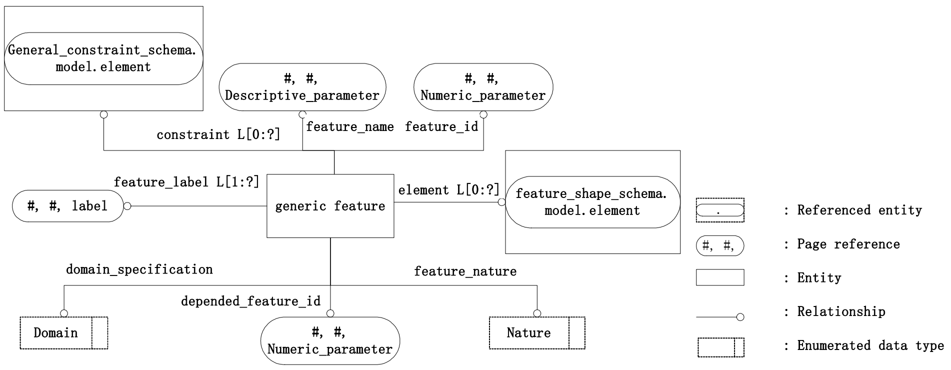


Fig. 1. Generic feature representation schema

2. GENERIC FEATURE MODEL

To consider integrating a solid modeler with the feature-oriented database, the mapping method between the database schemas and the feature definitions to the solid modeler entities is critical. Unified feature model allows different applications to define different features with a set of well-defined generic types [5], because it is essential that each feature type has well-defined semantics [2]. The semantic attributes specified in each feature definition have to be associated with the structured elements of the given feature type. Such elements include feature shape representation with parameters, constraints that all feature instances should satisfy, and the non-geometric attributes to be used for embedded semantic properties, such as classifications, names, labels, and relations. All types of constraints are used for capturing design intent in the context of product shape models. A generic feature representation schema can be described in EXPRESS-G as shown in Fig.1.

2.1 Feature Shape Representation

To represent the shape of a feature means defining feature geometrical and topological constraints or relations with parameters and associating these parameters with feature manipulation (creation, modification and deletion) functions. The parameters are used to provide user interfaces to create and modify features in the modeling operations. The database schemas for geometric entities have been reported in [12].

2.2 Constraint Definition

Constraints must be explicitly defined in the feature model to specify relationships among features, geometric or topological entities, and to provide invariant characteristics in the model. Constraints may have various types (e.g. geometric constraints, tolerance constraints and others). In a feature definition, constraints are regarded as attributes attached to a set of associated entities, e.g. geometric and non-geometric entities or even features. Although different types of constraints may have different attributes, but they fall into a few common types, which can be generalized as shown in Fig. 2.

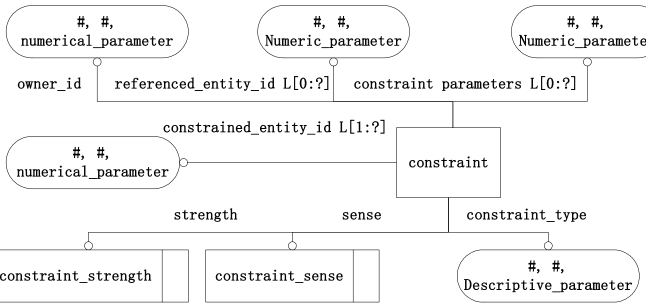


Fig. 2. Constraint representation schema

1) *Owner_ID.* It uniquely identifies which feature the constraint belongs to.
2) *Constraint_type.* Constraint may have various types (e.g. geometric constraint, tolerance constraint and others). Different kinds of constraint type relate to different constraint solving functions.
3) *Constraint_parameter.* It specifies the value of a constraint's evaluation functions, e.g. dimensions.
4) *Referenced_entity_ID list.* It is a list of pointers of referenced entities, which are used to evaluate the relations with those constrained entities. Note, hereafter, geometric entities refer to features, solids, faces, surfaces, edges, curves, vertices, points, datum geometries, etc. collectively.
5) *Constrained_entity_ID list.* It specifies a list of pointers of entities whose properties are controlled and evaluated by the constraint methods with reference to the referenced entities.
6) *Constraint_strength.* It is an enumeration data, which may have several levels, such as required, strong, medium or weak. It represents the extent that the constraint needs to be imposed when a constraint conflict among a few of them.
7) *Constraint_sense.* It is used to specify the constraint nature, data structure, and the directions between constrained entities and referenced entities. It has the select data type as defined in EXPRESS which maybe directed or undirected. A constraint is directed if there is not any closed reference loop among all its constrained entities. A constraint is undirected if there are crossed control relations among constrained entities and form a closed reference loop among them. For example, if a directed constraint is applied to two lines (line_1 and line_2) and it

requires line_2 parallels with reference to line_1, it implies that line_1 has been existed in the model before line_2 was created. The corresponding undirected constraint would simply assert that line_1 and line_2 are parallel, with no implied precedence in their order of creation.

8) *Constraint solving functions.* They are responsible for solving constraint conditions and validations according to the constraint's type.
9) *Other manipulation functions.* These functions may include attributes access functions, behavior control functions and so on.

2.3 Other Feature Properties

Other feature properties can be defined as follows:

- General feature attributes- Feature_name and feature_id

General feature attributes such as feature_name and feature_id shall be realized with instantiation of a specific feature according to the application_specific feature definition. These attributes serve as the necessary attributes for searching of feature during feature modeling operations.

- Depended_feature_id_list

To maintain feature relationship, depended_feature shall be explicitly defined in the feature definition. Feature dependency relation definition is described by Biddara [2, 3] as "feature *f1* directly depends on feature *f2* whenever *f1* is attached, positioned or, in some other way, constrained relative to *f2*". Depended_feature_id_list plays an important role in maintaining feature dependency graph, and furthermore, feature relations during feature modeling operations.

- Feature label

A feature label is attached as an attribute to every face of a particular feature instance. In a feature, its member face labels are defined as a list of strings in the definition, to record feature face elements. Then the face corresponding to the label is referred as the owner.

- Domain specification

Domain specification has the ENUMERATION data type, which represents the application scope such as *design*, *manufacturing*, *assembly* and others. By specifying the different domains, multi-views can be supported with certain filtering and synchronizing mechanisms.

- Nature

The nature of a feature has ENUMERATION data type as well which is either positive or negative. A positive value means the instances of the feature are created by adding material. A negative value means forming a feature instance is realized by subtracting material.

2.4 Member Functions

- Attributes access functions

Attribute access functions shall be defined to manage feature's attributes. Some functions are common to all types of features, e.g. *backup()*. Others are feature-specific such as *findOwner()*, *findConstraint()*, *getParameter()*, *setParameter()* and so on. Object technology with a proper polymorphism design can be applied well here.

- Modeling operation functions

Modeling operation functions (e.g. *splitOwner()*, *mergeOwner()*) are used to control the behaviors of feature during a modeling operation, e.g. splitting, merging, or translation.

- Feature evaluation and validation functions

Feature evaluation functions are responsible for feature model modification. Feature validation functions are used to validate feature geometry and solving constraints after each feature modeling operation. These functions will be discussed in detail in section 4.

- Saving and restoring functions

In order to persistently manage product and process information, which includes feature information, geometrical data and other information, saving and restoring functions of the database, which are the interactions between the run-time feature model and the database, must be defined in individual feature classes because these functions have to organize information for different application views according to users' requirements. Details will be explained in section 4.

3. MAPPING MECHANISMS

To provide lower-level geometrical modeling services, a geometrical modeling kernel is required. In this work, ACIS, a commercial package, is incorporated into the proposed system. An EXPRESS-defined and extended STEP feature model, which includes geometrical and generic feature representation schemas, is mapped to the data representation schemas in ACIS such that the proposed system will have the required fine grain functionality. On the other hand, this feature model needs also to be mapped to the target database schema so that it can be interfaced with a persistent repository.

3.1 Mapping From Extended EXPRESS Model To ACIS Workform Format

3.1.1 Geometry mapping

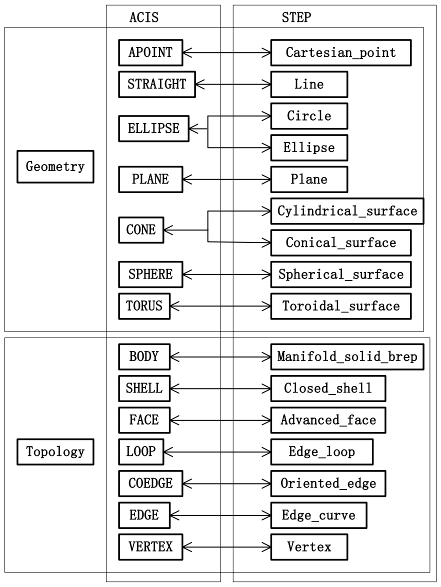


Fig. 3. Mapping of STEP entities and ACIS entities

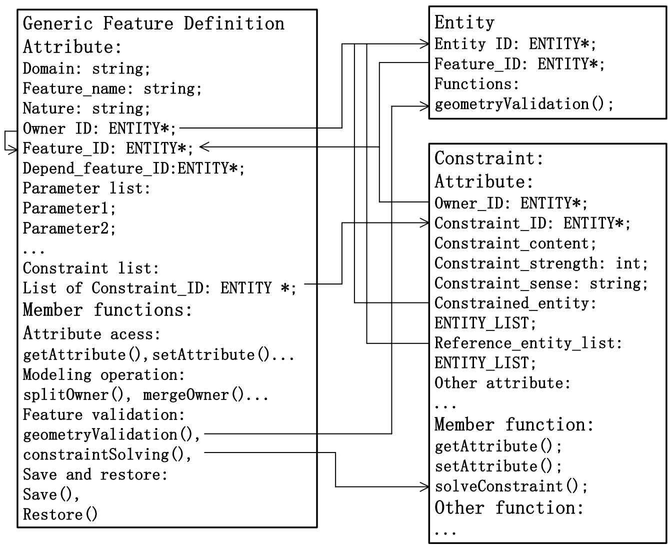


Fig. 4. Generic feature definition with ACIS entities

In this research, STEP part 42 [6] is adopted as the geometrical representation schema for all feature models. The geometry and topology entity mapping between ACIS and STEP has been developed as shown in Fig. 3.

3.1.2 Generic feature definition under ACIS framework

ACIS provides ENTITY-ATTRIBUTE architecture [1], under which we can define user-defined attributes (features, constraints or others). The following rules are developed and used by the authors for defining features, constraints and other attributes in ACIS:

- Use simple attributes to represent properties such as the material of a body or color of a face.
- Use complex attributes to represent properties such as features, dimensions, tolerance, or constraints.
- Use bridging attributes to link an ENTITY with some application-specific, variable length data.
- Use instruction attributes placed on entities to force certain behavior.
- Attributes of features and constraints may have various data types, e.g. string, integer or ENTITY pointer.
- Aggregate data type shall be defined as ENTITY_LIST. The ENTITY_LIST is a variable length associative array of ENTITY pointers and provides common functions for the manipulation of its members, e.g. *add* ENTITY, *look up* ENTITY and *[]* operator for accessing list member by position.
- Enumeration data type can be simulated by defining a string as the enumeration member or simply using integer data type.
- Selecting data type can be simulated by using an abstract class and defining specific types of the abstract class.

On the basis of the above proposed mapping rules, a generic feature definition is created as shown in Fig.4.

3.2 Database Representation Schema

According to the mapping mechanisms proposed in [12], a geometrical representation schema as well as generic feature representation schema in the database has been developed. For details, please refer to [12].

4. THE INTEGRATION OF THE SOLID MODELER AND THE DATABASE

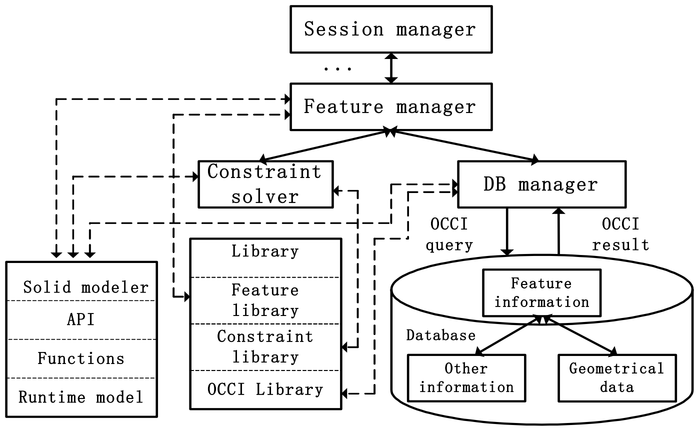


Fig. 5. Partial diagram for the integration of a solid modeler and the feature-oriented database

The solid modeler has been tightly integrated in four layers in order to manage product and process information (see Fig. 5.). First, its API functions are called constantly which are encapsulated within the feature manipulation methods during the collaboration sessions between the end users and the application server. Second, all the geometrical entities are manipulated and their run-time consistency maintained through the solid modeler's implicit runtime data structure module. Third, it also provides runtime functional support directly to the end users via commands dynamically. Fourth, the solid modeler has also to support the repository operations via the DB manager.

This paper focuses on the forth layer. In the proposed architecture of the web-based feature modeling system [12], database manager (DB manager) is responsible for managing the geometrical entities via the solid modeler runtime model and manipulating the data elements to be stored and extracted in the database for different applications. With the support of a solid modeler, the database manager can provide data manipulation functions such as save, restore and validate functions. These functions are fundamental to support different applications. In the following sub-sections, feature validation as well as the generic save and restore algorithms are explained. In order to manage the connection between the DB manager and the database during saving and restoring processes, OCCI (Oracle C++ Call Interface) [10] is adopted as the bridge (see Fig. 5.).

4.1 Feature Model Re-evaluation and Constraint Solving

Once feature operations are specified via User Interfaces (UIs), the product model needs to be modified and updated. This process is achieved through feature evaluation. The geometrical model has to be managed to ensure the consistency. Here, the run-time product model should be generated via the integrated solid modeler and managed based on the database records. All feature evaluation operations call solid modeler APIs to realize the geometrical procedures while the rest of the functions are implemented separately. In this way, the details of geometrical operations are readily looked after by the solid modeler; hence, the development effort is significantly reduced. Details for feature model re-evaluation will be explained in section 5.

Theoretically, feature validation functions include two kinds: those dealing with the geometry, and those dealing with constraints. With the incorporation of a solid modeler, geometry validation functions are not really necessary under the proposed design because the solid modeler is responsible for manipulating and validating feature geometry. On the other hand, constraint-solving functions need call specific algorithms defined in the individual constraint sub-classes to solve different kinds of constraints according to their types. Globally, all the constraints are maintained by the Constraint Manager in a constraint graph for EPM (Entire Product Model), which contains sub-graphs for specific application views. Constraint manager solves constraints by calling the corresponding solvers according to different constraint types. For example, SkyBlue algorithm [11] can be used to solve local algebraic constraints in design domain; Degrees of Freedom analysis algorithm [8] can be used to solve geometrical constraints in design domain. If conflict of intra-application constraints occurs, local constraints solver can determine automatically which constraint should be satisfy first according to the value of *constraint_strengh*, which is an attribute of constraint defined in section 2. Inter-application constraints can also be solved under the control of constraint manager according to the value of domain_strength. For definition of *domain_strength*, please refer to section 2. The value of *domain_strength* can be predefined which regulates priority sequence of different domains, or is set by an authorized user. Any conflict of inter-application constraints will be detected by constraint manager after which the constraints solver can trigger corresponding applications to reevaluate the product model according to *domain_strength*. Only when all constraints are checked and feature geometry is validated, does feature validation finish.

4.2 *Save* Algorithm

To elaborate, during the saving process, the solid modeler has to extract all the information from its runtime data structure and then save them into the database after a format conversion according to the mapping relations as shown in Fig. 4. and the database mapping schema described in [12]. The Save algorithm can be expressed in the steps as follows (see Fig. 6.):

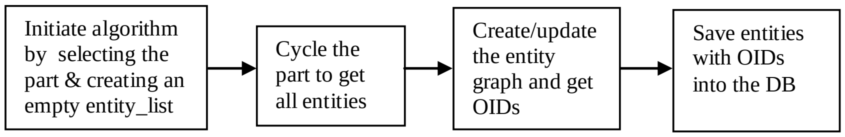

Initiate algorithm by selecting the part & creating an empty entity_list
Cycle the part to get all entities
Create/update the entity graph and get OIDs
Save entities with OIDs into the DB

Fig. 6. *Save* algorithm

1) Select the part to be saved. Create an empty entity list and add the part attributes to be saved to the list;
2) Cycle all entities (features, topological entities, such as solids, shells, faces, and geometrical entities, such as lines, planes, curves, and surfaces) from the part and add them to a graph map so that object pointers can be fixed as unique database Object Identifiers (OID). ACIS API functions, e.g. *api_get_xxxx()*, are used to get all saved ENTITIES;
3) Use such object pointers to call *save* functions of the specific class (e.g. *point.save()*, *vertex.save()* or *feature.save()*) to save part data to the database.

4.3 *Restore* Algorithm

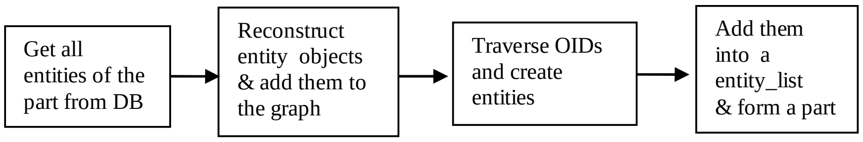

Get all entities of the part from DB
Reconstruct entity objects & add them to the graph
Traverse OIDs and create entities
Add them into a entity_list & form a part

Fig. 7. *Restore* algorithm

In a reverse way, the uploading process is triggered when the product model is being established during the session initiation from the database.

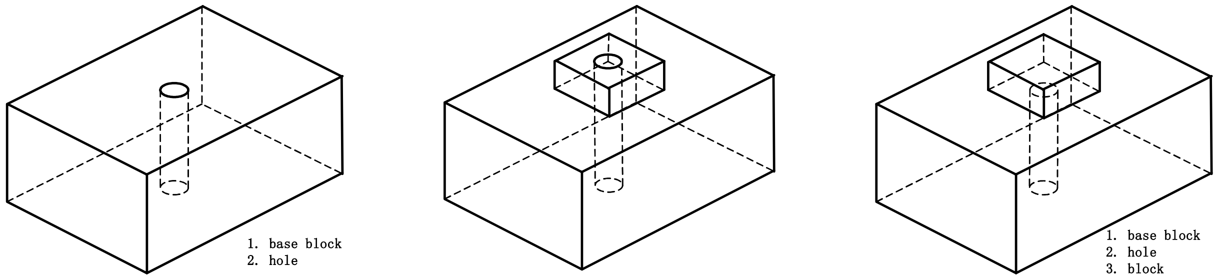
Restore algorithm has the following steps (Fig. 7.):

1) All the entities of a part are retrieved from the database by searching their linked Object Identifiers (OIDs);
2) Reconstruct new objects, e.g. features, geometrical entities, topological entities. Upon reconstruction, all the objects will be validated;
3) Add all the entities to a newly generated object graph map;
4) Convert these OIDs to genuine pointers;
5) Create an entity list and add all the entities to the list to form the part. Validation, e.g. geometry and feature validation will be carried out during this procedure.

5. FEATURE MODEL RE-EVALUATION

5.1 Problems Of Historical-dependent System

For most parametric and history-based modeling system, feature model is re-evaluated by re-executing whole or part of the model history. The disadvantages of this method are the high computational cost and the considerable amount of storage space [2]. Moreover, history-based model re-evaluation causes ambiguous feature semantics due to the static chronological feature creation order in the model history. This is illustrated in the example shown in Fig. 8. The simple part consists of a base block and a through hole. Later on, the designer wants to modify the part by adding another block and extending the depth of hole so that he can get expected part model as shown in Fig. 8(b). However, sometimes unexpected modeling results can be generated by the history-based reevaluation of the model as shown in Fig. 8(c)., because the feature creation order is *baseblock->hole->block*. In order to get the expected part model, the precedence order, in this example, should be changed to *baseblock->block->hole*. This semantic problem is caused by the static precedence order in the model history on which model re-evaluation is based. From this example, it is clear that the precedence relation among features should be dynamically maintained and updated after each modeling operation.



(a) Example part at the initial state (b) Expected result after modification (c) Result of history-based re-evaluation after modification

Fig. 8. Semantic problem for historical-dependent system

5.2 Dynamically Maintaining Feature Precedence Order

In this work, feature precedence order is maintained dynamically based on a feature dependency graph. Feature dependency relations are explicitly defined in the feature definition as explained in section 3. The followings are the rules for feature precedence determination.

For dependent features:

If feature f1 depends on feature f2, then f2 precedes f1.

Relations between independent features can be determined by feature overlapping detection. Following is the rule for feature precedence determination among independent features:

For two features with different natures, if they overlap with each other, the feature with positive nature always precedes the feature with negative nature.

According to the above rules for feature precedence determination, an algorithm is proposed to dynamically maintain feature precedence order after each feature modeling operation.

1) Find all the features of the part and add them to a graph map (unsorted);

2) Partially sort the graph map according the feature dependency graph. This is done by the algorithms described in Fig. 9.;
3) Sort the partially sorted graph with reference to overlap detection result.

In this way, a global feature precedence order can be updated dynamically after each feature modeling operation.

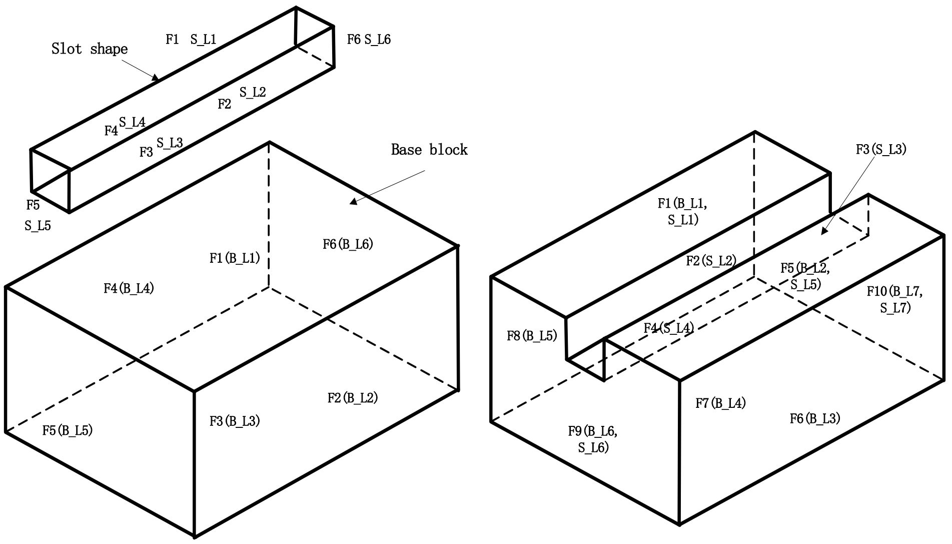
```
(For i=1; i<n-1; i++)
{ (for j=i+1, j<n; j++)
   {if (Pxj>Pxi)
     {Xm= Xi;
      Xi=Xj;
      Xj=Xm;
      }
   }
}
```

Here:
X_i represents any feature in the feature set;
X_j represents depended feature of X_i;
Px_i represents the position of feature X_i in the feature map;

Fig. 9. Algorithm for precedence order generation

5.3 History-independent Feature Model Re-evaluation

Based on the proposed global feature order generating algorithms, the re-evaluation of the feature model after modification can be described as follows:



F_i : ith face; S_L_i : ith slot_label; B_L_i : ith block_label.

Fig. 10. Feature creation

5.3.1 Adding a new feature instance

This is carried out as follows:

1) Create the shape of the new feature according to feature creation schema;
2) Attach labels to every face of the feature instance; and
3) Carry out Boolean operation according to the nature of feature (for positive nature, Boolean union operation is carried out; for negative nature, Boolean subtraction is done).

Feature labels must be properly propagated during the Boolean operation because labels are attached to every face of the feature instance. For those feature faces to be kept after Boolean operation and with no overlapping to the base block faces, their feature labels are carried into the block. The tricky issue is how to deal with overlapping faces. This process is controlled in the splitOwner(), mergeOwner() functions defined in FEATURE_LABEL class. The geometric relation between a pair of overlapping faces – one from the new feature and the other from the base block - i.e. the choice that either splitOwner() or mergeOwner() is to be used, is determined by the Boolean operation algorithms. The followings are the rules for controlling feature label propagation for overlapping faces during a Boolean operation:

1) If the owner (face) of a feature label is going to be split, the label should also be propagated to the new faces;
2) If the owner (face) of a feature label is going to be merged with an existing face that belongs to the base block and the owner will be deleted, the label should be propagated to the existing block face. In this case the new feature depends on the existing block face.

Fig. 10. illustrates the creation of a slot feature on the base block. The shape of the slot is first created and labels (S_L) are attached to every face of the slot shape. Then Boolean subtraction is carried out as the nature of slot is negative. During Boolean operation, all faces of the original slot shape are deleted and three new faces (F2, F3, F4) are generated. The top face of original base block is split into two faces (F1 and F5). According to the above rules, slot feature labels are attached to three new faces, as well as the two top faces and two side faces of the base block.

5.3.2 Deleting a feature instance

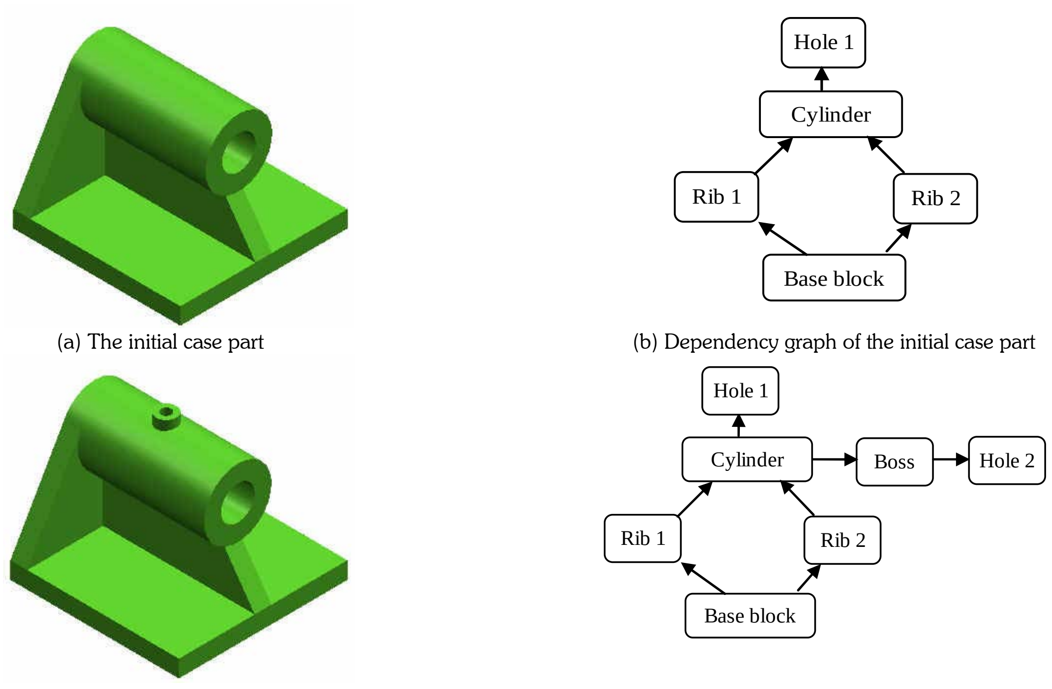
This is carried out as follows (assuming no other feature depends on the feature to be deleted):

1) Find all faces labeled only as elements of this feature instance;
2) Delete these faces and their associated topological and geometrical entities;
3) Merge the adjacent faces which belong to the same feature.

To delete the slot feature from the part in Fig. 10., all faces labeled as slot are found first. Then faces (F2, F3, F4) that contain only slot label are deleted. Finally, adjacent faces are merged and solid is repaired.

5.3.3 Modifying a feature instance

To modify a feature instance, we shall first check which features that are involved (to be modified) in the modeling operation. This checking is based on the feature dependency graph and global feature precedence order. Then, features involved will be removed from the product model first. Then these independent features are updated according to the operational requirements. Finally, these modified features are inserted back to the model.



(a) The initial case part

(b) Dependency graph of the initial case part

(c) After adding a boss and a hole

(d) Dependency graph for the modified part

Fig. 11. Case study model for re-evaluation

6. CASE STUDY

The proposed feature-oriented database has been implemented coupled with a geometrical modeling kernel, ACIS. Design features and constraints have been defined and some example parts have been tested. Fig. 11(a). shows an example part made up of a base block, two ribs and a hole. Fig. 11(b). shows its feature dependency graph. Fig. 11(c). is the result of adding another boss and a hole on the cylinder. Fig. 11(d). gives the new dependency graph after the operation. Based on the feature dependency graph, the removal of boss feature can be done by removing the boss as well as the hole feature that depends on the boss feature.

7. CONCLUSION

In this paper, the integration of a feature-oriented database and a solid modeler is presented. The mapping mechanisms, from EXPRESS-defined generic feature model entities to ACIS workform format, and the integration with the repository database schema are described. Generic algorithms for feature manipulation with the solid molder and database methods are also illustrated. Based on the working prototype system, it can be concluded that solid modeler can be effectively integrated with the feature-oriented database to provide lower-level geometrical modeling services. This kind of integration can further enable information sharing among different applications and Web enabled engineering collaboration. Finally, a modeler-supported, history-independent feature model re-evaluation approach is described in detail.

8. REFERENCES

[1] ACIS Online Help User's Guide. Available at http://www.spatial.com.

[2] Bidarra, R., Bronsvoort, W.F., "Semantic feature modeling", *Computer-Aided Design* Vol. 32, pp. 201–225, 2000.

[3] Bidarra, R., van den Berg, E. and Bronsvoort, W. F., "Collaborative modeling with features", *Proceedings of DETC'01 2001 ASME Design Engineering Technical Conferences* September 9-12, 2001, Pittsburgh, Pennsylvania.

[4] Bronsvoort, W. F., Bidarra, R. and Noort, A., "Semantic and multiple-view feature modeling: towards more meaningful product modeling". In: *Geometric Modelling - Theoretical and Computational Basis towards Advanced CAD Applications,* edited by Kimura, F., Kluwer Academic Publishers, Dordrecht, 69-84, 2001.

[5] Chen, G., Ma, Y. S., Thimm, G. and Tang, S. H., "Unified feature modeling scheme for the integration of CAD and CAx", *Computer-Aided Design & Applications*, Vol. 1, Nos. 1-4, 2004, *CAD'04*, pp 595-602.

[6] ISO 10303, Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 42: *Integrated Generic Resources: Geometric and Topological Representation*, ISO 10303-42:1994 (E), ISO, Geneva, 1994.

[7] Kim, J. and Han, S., "Encapsulation of geometric functions for ship structural CAD using a STEP database as native storage", *Computer-Aided Design*, Vol. 35, pp. 1161–1170, 2003.

[8] Kramer, G. A. "Solving geometric constraints systems: a case study in kinematics", *The MIT Press*, Cambridge, MA, USA, 1992.

[9] Martino, T. D., Falcidieno, B. and Habinger, S., "Design and engineering process integration through a multiple view intermediate modeler in a distributed object-oriented system environment", *Computer-Aided Design*, Vol. 30, No. 6, pp. 437-452, 1998.

[10] ORACLE online documentation. Available at http://www.oracle.com.

[11] Sannella, M. "The SkyBlue constraint solver and its applications", *First Workshop on Principles and Practice of Constraint Programming*, 1993.

[12] Tang, S. H., Ma, Y. S. and Chen, G., "A feature-oriented database framework for web-based CAx applications", *Computer-Aided Design & Applications*, Vol. 1, Nos. 1-4, 2004, *CAD'04*, pp 117-125.